Algoritmi i strukture podataka
11. Čas — Pretraga

Sana Stojanović Đurđević

December 21, 2018

Pred vama je prateći materijal koji se može koristiti uz skriptu. Sastoji se od rešenja nekoliko zadataka koji su pređeni na vežbama i predavanjima.

1 Generisanje kombinatornih objekata

1.1 Svi podskupovi

Problem: Napiši funkciju koja nabraja i obrađuje sve podskupove datog skupa čiji su elementi predstavljeni datim vektorom.

Napomena: Iako jezik C++, kao i mnogi drugi savremeni jezici, pruža tip za reprezentovanje skupova, implementacija je jednostavnija i efikasnija ako se elementi skupa čuvaju u nizu ili vektoru.

Jedan način rešavanja ovog problema je induktivno rekurzivni, i on je detaljno objašnjen u skripti sa predavanja. Ovde će biti prikazano rešenje koje koristi funkciju koja na osnovu datog objekta generiše sledeći objekat (u leksikografskom poretku).

Svi podskupovi skupa brojeva $\{1, 2, 3, 4\}$ se mogu grupisati na naredni način:

```
- 1  12  123  1234
        124
     13  134
     14
  2  23  234
     24
  3  34
  4
```

Možemo da primetimo da se *naredni* skup može dobiti od tekućeg na jedan od naredna dva načina:

1. Prvi način je *proširivanje* — kada se naredni podskup dobija dodavanjem nekog elementa u prethodni (koraci u prethodnoj tabeli kod kojih se prelazi u narednu kolonu, na desno). Da bi dobijeni podskup sledio

neposredno iza prethodnog u leksikografskom redosledu, dodati element podskupu mora biti najmanji mogući. Pošto je svaki podskup sortiran, element mora biti za jedan veći od poslednjeg elementa podskupa koji se proširuje (izuzetak je prazan skup, koji se proširuje elementom 1). Jedini slučaj kada proširivanje nije moguće je kada je poslednji element podskupa najveći mogući (u našem primeru to je 4).

2. Drugi način je *skraćivanje* kada se naredni element dobija uklanjanjem nekih elemenata iz podskupa i izmenom preostalih elemenata. To su koraci u prethodnoj tabeli kod kojih se prelazi sa kraja jedne u narednu vrstu. U ovom slučaju skraćivanje funkcionie tako to se iz podskupa izbaci završni najveći element, a zatim se najveći od preostalih elemenata uveća za 1 (on ne može biti najveći, jer su elementi unutar svake kombinacije strogo rastući). Ako nakon izbacivanja najvećeg elementa ostane prazan skup, naredna kombinacija ne postoji.

```
#include <iostream>
#include <vector>

using namespace std;

void obradi(vector<int> podskup){
  for(int i=0; i<podskup.size(); i++)
    cout << podskup[i];
  cout << endl;
}
```

```
// na osnovu datog podskupa skupa {1, ..., n} odredjuje
// leksikografski naredni podskup i vraca da li takav
// podskup postoji
bool sledeciPodskup(vector<int>& podskup, int n) {
  // specijalni slucaj prosirivanja praznog skupa
  if (podskup.empty()) {
    podskup.push_back(1);
    // podskup je uspesno pronadjen
    return true;
  }
  // prosirivanje
  if (podskup.back() < n) {
    // u podskup dodajemo element koji je za 1 veci od
    // trenutno najveceg elementa
    podskup.push_back(podskup.back() + 1);
    // podskup je uspesno pronadjen
    return true;
  }
  // skracivanje
  // uklanjamo poslednji najveci element
  podskup.pop_back();
  // ako nema preostalih elemenata ne postoji naredni podskup
  if (podskup.empty())
    return false;
  // najveci od preostalih elemenata uvecavamo za 1
  podskup.back()++;
  // podskup je uspesno pronadjen
  return true;
}
```

```
// obrada svih podskupova skupa {1, ..., n}
void obradiPodskupove(int n) {
  // tekuci podskup
  vector<int> podskup;
  // obradjujemo podskupove redom, sve dok je moguce pronaci
  // leksikografski sledeci podskup
  do {
    obradi(podskup);
  } while (sledeciPodskup(podskup, n));
}
```

```
int main(){
  int n;
  cout << "Unesite broj n:" << endl;
  cin >> n;

  obradiPodskupove(n);
}
```

1.2 Sve varijacije

Problem: Definisati funkciju koja obrađuje sve varijacije sa ponavljanjem dužine k skupa $\{1, \ldots, n\}$.[1]

Prikazaćemo rešenje koje kreira narednu varijaciju u odnosu na leksikografski redosled. Na primer, ako nabrajamo varijacije skupa $\{1, 2, 3\}$ dužine 5 naredna varijacija za varijaciju 21332 je 21333, dok je njoj naredna varijacija 22111. Varijacija 33333 nema leksikografski sledeću varijaciju.

Može se primetiti da se izvršavaju dve *akcije*: (1) uvećanje poslednjeg broja u varijaciji koji nema maksimalnu vrednost (u našem primeru to je vrednost 3) i (2) postavljanje svih brojeva desno od njega na minimalnu vrednost (u našem primeru to je vrednost 1).

Pozicija na kojoj se nalazi broj koji se uvećava naziva se prelomna tačka (engl. turning point).

U našem primeru, prelomna tačka za varijaciju 21332 je pozicija 4 (poslednja pozicija u nizu), i naredna varijacija je 21333. Za tu varijaciju prelomna tačka je pozicija 1 na kojoj se nalazi element 1 i naredna varijacija je 22111. Niz 33333 nema prelomnu tačku, pa samim tim ni leksikografski sledeću varijaciju.

Implementacija: Varijaciju obilazimo od kraja postavljajući na 1 (na minimalnu vrednost) svaki element u varijaciji koji je jednak broju n (maksimalnoj vrednosti). Ako se zaustavimo pre nego što smo stigli do kraja niza, znači da smo pronašli prelomnu tačku, element koji se može uvećati i uvećavamo ga. U suprotnom je varijacija imala sve elemente jednake n i bila je maksimalna u leksikografskom redosledu.

[1] Podsećanje: Varijacija bez ponavljanja dužine k u skupu od n elemenata ima $n(n-1)\ldots(n-k+1)$. Varijacija sa ponavljanjem dužine k u skupu od n elemenata ima n^k.

```
#include <iostream>
#include <vector>

using namespace std;

void obradi(vector<int> varijacija){
  for(int i=0; i<varijacija.size(); i++)
    cout << varijacija[i];
  cout << endl;
}

bool sledecaVarijacija(int n, vector<int>& varijacija) {
  // od kraja varijacije trazimo prvi element koji se moze povecati
  int i;
  int k = varijacija.size();
  for (i = k-1; i >= 0 && varijacija[i] == n; i--)
    varijacija[i] = 1;
  // svi elementi su jednaki n - ne postoji naredna varijacija
  if (i < 0)
    return false;

  // uvecavamo element koji je moguce uvecati
  varijacija[i]++;
  return true;
}
```

```
void obradiSveVarijacije(int k, int n) {
  // krecemo od varijacije 11...11 - ona je leksikografski najmanja
  vector<int> varijacija(k, 1);
  // obradjujemo redom varijacije dok god postoji leksikografski
  // sledeca
  do {
    obradi(varijacija);
  } while (sledecaVarijacija(n, varijacija));
}
```

```
int main(){
  int k, n;
  cout << "Unesite n i k: " << endl;
  cin >> n >> k;

  obradiSveVarijacije(k,n);
  return 1;
}
```

1.3 Svi binarni zapisi bez uzastopnih jedinica

Problem: Definisati funkciju koja obrađuje sve binarne zapise dužine n kojima se ne javljaju dve uzastopne jedinice.

Obradićemo slučaj kada generišemu sledeći zapis na osnovu prethodnog (u leksigorafskom redosledu).

Ovaj algoritam će predstavljati modifikaciju algoritma kojim se određuje sledeća varijacija u leksikografskom redosledu. Prethodni primer je urađen za skupove oblika $\{1, 2, \ldots, n\}$, a sada nam treba varijacija nad skupom $\{0, 1\}$ pa je najmanja vrednost jednaka 0, a najveća vrednost jednaka 1.

Krećemo sa kraja niza i upisujemo nule sve dok se na trenutnoj ili na prethodnoj poziciji u nizu nalazi jedinica. Na kraju, na poziciji na kojoj smo se zaustavili i nismo upisali nulu (ako takva postoji) upisujemo jedinicu (to je pozicija na kojoj piše nula i ispred nje ili nema nita ili piše nula). Ako takva pozicija ne postoji, onda je trenutni niz leksikografski najveći.

```
#include <iostream>
#include <vector>

using namespace std;

bool sledecaVarijacijaBezUzastopnihJedinica(string& s){
  int n = s.length();
  int i = n - 1;
  while ((i >= 0 && s[i] == '1') || (i > 0 && s[i-1] == '1'))
    s[i--] = '0';
  if (i < 0)
    return false;
  s[i] = '1';
  return true;
}

int main(){
  int n;
  cout << "Unesite n: " << endl;
  cin >> n;
  string s(n,'0');

  do {
    cout << s << endl;
  } while (sledecaVarijacijaBezUzastopnihJedinica(s));

  return 1;
}
```

1.4 Sve permutacije

Problem: Definisati proceduru koja nabraja i obrađuje sve permutacije skupa $\{1, 2, \ldots, n\}$. Na primer, za n = 3, permutacije su 123,132,213,231,312 i 321. [2]

Rekurzivno generisanje permutacija u leksikografskom redosledu je veoma komplikovano, tako da ćemo se odreći uslova da permutacije moraju biti poređane leksikografski.

Implementiraćemo funkciju koja pronalazi narednu permutaciju u leksikografskom poretku. Razmotrimo permutaciju 13542. Pošto je niz 542 strogo opadajući, to nam govori da nije moguće ni na koji način razmeniti ta tri elementa da se dobije leksikografski veća permutacija, tj. ovo je najveća permutacija koja počinje sa 13. Dakle, naredna permutacija će biti leksikografski najmanja permutacija koja počinje sa 14, a to je 14235.

```
#include <iostream>
#include <vector>

using namespace std;

void obradi(vector<int> permutacija){
  for(int i=0; i<permutacija.size(); i++)
    cout << permutacija[i];
  cout << endl;
}
```

[2] Permutacija bez ponavljanja dužine n ima $n!$.

```
bool sledecaPermutacija(vector<int>& permutacija){
  int n = permutacija.size();
  // linearnom pretragom pronalazimo prvu poziciju i takvu da
  // je permutacija[i] > permutacija[i+1]
  int i = n - 2;
  while (i >= 0 && permutacija[i] > permutacija[i+1])
    i--;
  // ako takve pozicije nema, permutacija je leksikografski maksimalna
  if (i < 0)
    return false;
  // linearnom pretragom pronalazimo prvu poziciju j takvu da
  // je permutacija[j] > permutacija[i]
  int j = n - 1;
  while (permutacija[j] < permutacija[i])
    j--;
  // razmenjujemo elemente na pozicijama i i j
  swap(permutacija[i], permutacija[j]);
  // obrcemo deo niza od pozicije i+1 do kraja
  for(j = n - 1, i++; i < j; i++, j--)
    swap(permutacija[i], permutacija[j]);
  return true;
}
```

```
int main(){
  int n;
  cout << "Unesite n: " << endl;
  cin >> n;

  vector<int> permutacija(n);

  for(int i=0; i<n; i++)
    permutacija[i] = i+1;

  do {
    obradi(permutacija);
  } while(sledecaPermutacija(permutacija));

  return 1;
}
```

2 Iscrpna pretraga (gruba sila)

2.1 Provera da li je formula tautologija

Gruba sila podrazumeva generisanje istinitosne tablice, izračunavanje vrednosti formule u svakoj vrsti (valuaciji) i proveri da li je formula u toj valuaciji tačna.

Valuacije predstavljaju varijacije dužine n dvočlanog skupa tačno, netačno, gde je n ukupan broj promenljivih.

Prikažimo jedno moguće rešenje. Pretpostavićemo da je formula predstavljena binarnim drvetom. Razlikuju se dva tipa čvora: listovi u kojima se nalaze celobrojne vrednosti i unutrašnji čvorovi u kojima se nalaze operatori. Pretpostavićemo i da je dat pokazivač na koren, kao i funkcija izračunavanja vrednosti formule za datu valuaciju.

Definisaćemo jedan strukturni tip u kojem objedinjavamo podatke koji se koriste i u listovima i u unutrašnjim čvorovima. Da ne bismo brinuli o dealokaciji drveta, umesto običnih pokazivača, koristićemo pametne pokazivače koje imamo na raspolaganju u jeziku C++.

shared_ptr tip podataka je pametni tip pokazivača koji se koristi kada je potrebno da više različitih vlasnika mogu da pristupaju istoj lokaciji u memoriji. Svi vlasnici imaju pristup kontrolnom bloku koji među ostalom čuva i ukupan broj vlasnika koji pokazuju na njega. Kada se ukupan broj vlasnika smanji na nulu, kontrolni blok uklanja podatak iz memorije i sebe samog.

`http://www.cplusplus.com/reference/memory/shared_ptr/`

Primer upotrebe:

`https://docs.microsoft.com/en-us/cpp/cpp/how-to-create-and-use-shared-ptr-instances?view=vs-2017`

use_count funkcija vraća broj vlasnika koji dele pristup nad istim objektom. Ako je u pitanju prazan pokazivač, funkcija vraća nulu.

`http://www.cplusplus.com/reference/memory/shared_ptr/use_count/`

make_shared funkcija alocira u memoriji i kreira objekat određenog tipa. Povratna vrednost ove funkcije je tipa `shared_ptr` koja čuva pokazivač na taj objekat i postavlja broj vlasnika (`use_count`) na 1.

`http://www.cplusplus.com/reference/memory/make_shared/`

U narednom delu koda se definiše reprezentacija operatora i pomoćnih funkcija za baratanje operatorima.

```
#include <iostream>
#include <vector>
#include <memory>

using namespace std;

enum TipCvora {PROM, I, ILI, NE, EKVIV, IMPL};

struct Cvor {
  TipCvora tip;
  int promenljiva;
  shared_ptr<Cvor> op1, op2;
};

typedef shared_ptr<Cvor> CvorPtr;

/* Funkcije za kreiranje cvorova */
CvorPtr NapraviCvor() {
  return make_shared<Cvor>();
}

/* Kreiranje cvora koji sadrzi promenljivu */
CvorPtr Prom(int p) {
  CvorPtr c = NapraviCvor();
  c->tip = PROM;
  c->promenljiva = p;
  return c;
}

/* Kreiranje cvora koji sadrzi operator */
CvorPtr Operator(TipCvora tip, CvorPtr op1, CvorPtr op2) {
  CvorPtr c = NapraviCvor();
  c->tip = tip;
  c->op1 = op1; c->op2 = op2;
  return c;
}
```

```
/* Pomocne funkcije za lakse kreiranje cvorova */
CvorPtr Ili(CvorPtr op1, CvorPtr op2) {
  return Operator(ILI, op1, op2);
}

CvorPtr Ii(CvorPtr op1, CvorPtr op2) {
  return Operator(I, op1, op2);
}

CvorPtr Ekviv(CvorPtr op1, CvorPtr op2) {
  return Operator(EKVIV, op1, op2);
}

CvorPtr Impl(CvorPtr op1, CvorPtr op2) {
  return Operator(IMPL, op1, op2);
}

CvorPtr Ne(CvorPtr op1) {
  return Operator(NE, op1, op1);
}
```

```
/* Izracunavanje vrednosti formule */
bool vrednost(CvorPtr c, const vector<bool>& valuacija) {
  switch (c->tip) {
  case PROM:
    return valuacija[c->promenljiva];
  case I:
    return
      vrednost(c->op1, valuacija) && vrednost(c->op2, valuacija);
  case ILI:
    return
      vrednost(c->op1, valuacija) || vrednost(c->op2, valuacija);
  case EKVIV:
    return
      vrednost(c->op1, valuacija) == vrednost(c->op2, valuacija);
  case IMPL:
    return
      !vrednost(c->op1, valuacija) || vrednost(c->op2, valuacija);
  case NE:
    return !vrednost(c->op1, valuacija);
  }
}
```

```
/* Pronalazenje sledece varijacije u leksikografskom poretku */
bool sledecaValuacija(vector<bool>& valuacija) {
  int i;
  for (i = valuacija.size() - 1; i >= 0 && valuacija[i]; i--)
    valuacija[i] = false;
  if (i < 0)
    return false;
  valuacija[i] = true;
  return true;
}
```

```
/* Da bi smo znali koliko elemenata treba da ima valuacija
   pronalazimo najvecu promenljivu koja se javlja u formuli */
int najvecaPromenljiva(CvorPtr formula) {
  if (formula->op1 == 0)
    return formula->promenljiva;
  int prom = najvecaPromenljiva(formula->op1);
  if (formula->op2 != nullptr) {
    int prom2 = najvecaPromenljiva(formula->op2);
    prom = max(prom, prom2);
  }
  return prom;
}

/* Pocetna valuacija je kada su sve promenljive netacne */
bool tautologija(CvorPtr formula) {
  vector<bool> valuacija(najvecaPromenljiva(formula) + 1, false);

  bool jeste = true;
  do {
    if (!vrednost(formula, valuacija))
      jeste = false;
  } while(jeste && sledecaValuacija(valuacija));

  return jeste;
}
```

```
/* Testiranje ovih funkcija u glavnom programu je jednostavno.*/

int main() {
  CvorPtr p = Prom(0);
  CvorPtr q = Prom(1);
  CvorPtr r = Prom(2);
  //CvorPtr formula = Ekviv(Ne(Ili(p, q)), Ii(Ne(p), Ne(q)));
  //CvorPtr formula = Impl(p, Impl(q,p));
  CvorPtr formula = Impl(Ii(Impl(p,q),Impl(q,r)),Ili(Ne(p),r));
  cout << "Formula " << (tautologija(formula) ? "jeste" : "nije");
  cout << " tautologija" << endl;
  return 0;
}
```

3 Pretraga sa povratkom (backtracking)

Algoritam pretrage sa povratkom (engl. backtracking) poboljšava tehniku grube sile tako što tokom implicitnog DFS obilaska drveta kojim se predstavlja prostor potencijalnih rešenja odseca one delove drveta za koje se unapred može utvrditi da ne sadrže ni jedno rešenje problema. Dakle, umesto da se čeka da se tokom pretrage stigne do lista drveta i da se provera vrši tek tada, prilikom pretrage sa povratkom provera se vrši u svakom koraku i vrši se provera parcijalno popunjenih torki rešenja. Kvalitet rešenja zasnovanog na ovom obliku pretrage uveliko zavisi od kvaliteta funkcije kojom se vrši odsecanje.

Funkcija odsecanja mora biti potpuno precizna u svim čvorovima koji predstavljaju kandidate za rešenje i u tim čvorovima mora potpuno precizno odgovoriti da li je tekući kandidat zaista ispravno rešenje. Dodatno, ta funkcija procenjuje da li se trenutna torka može proširiti do ispravnog rešenja. U tom pogledu funkcija odsecanja ne mora biti potpuno precizna: moguće je da se odsecanje ne izvrši iako se torka ne može proširiti do ispravnog rešenja, međutim, ako se odsecanje izvrši, moramo biti apsolutno sigurni da se u odsečenom delu zaista ne nalazi ni jedno ispravno rešenje. Ako je funkcija odsecanja takva da odsecanje ne vrši nikada, bektreking algoritam se svodi na algoritam grube sile.

3.1 Izlazak iz lavirinta

Problem: Napisati program koji odre duje da li je u lavirintu moguće stići od starta do cilja. Lavirint je određen matricom karaktera (`x` označava zid kroz koji se ne može proći i matrica je ograničena sa četiri spoljna zida, `.` označava slobodna polja, `S` označava start, a `C` označava cilj). Sa svakog polja dozvoljeno je kretanje u četiri smera (gore, dole, levo i desno).

```
xxxxxxxxxxxxxxx
xS...........x
x.xxxxxxxxxxxx
```

```
x............x
xxxxxxxxxxxxx.x
x............x
x.xxxxxxx.xxxxx
x...........Cx
xxxxxxxxxxxxxxx
```

Zadatak reavamo iscrpnom pretragom svih mogućih putanja. Pretragu možemo organizovati *u dubinu*. Funkcija prima startno i ciljno polje, pri čemu se startno polje menja tokom rekurzije. Ako je startno polje poklapa sa ciljnim, put je uspešno pronađen. U suprotnom, ispitujemo 4 suseda startnog polja i ako je susedno polje slobodno (nije zid), pretragu rekurzivno nastavljamo od njega (sudedno polje postaje novo startno polje). Potrebno je da obezbedimo da se već posećena startna polja ne obrađuju ponovo i za to koristimo pomoćnu matricu u kojoj za svako polje registrujemo da li je posećeno ili nije. Pre rekurzivnog proveravamo da li je susedno polje posećeno i ako jeste, rekurzivni poziv preskačemo, a ako nije označavamo da je to polje posećeno.

```
#include <iostream>
#include <vector>

using namespace std;

//typedef vector<vector<bool>> Matrica;
template<typename T>
using Matrica = vector<vector<T>>;


// funkcija koja kreira matricu postavljenu na false
Matrica<bool> napraviMatricu(int m, int n, bool value){

  Matrica<bool> mat;

  for(int i=0; i<m; i++){
    vector<bool> tekuce;
    for(int j=0; j<n; j++)
      tekuce.push_back(value);

    mat.push_back(tekuce);
  }

  return mat;
}
```

```
// ispituje da li postoji put kroz lavirint od polja (x1, y1) do
// (x2, y2) pri cemu matrica posecen oznacava koja su polja
// posecena, a koja nisu
bool pronadjiPut(Matrica<bool>& lavirint, Matrica<bool>& posecen,
 int x1, int y1, int x2, int y2) {
  // ako se pocetno i krajnje polje poklapaju, put postoji
  if (x1 == x2 && y1 == y2)
    return true;
  // cetiri smera u kojima se mozemo pomerati
  int pomeraj[4][2] = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};
  // pokusavamo pomeranja u svakom od tih smerova
  for (int i = 0; i < 4; i++) {
    // polje na koje se pomeramo
    int x = x1 + pomeraj[i][0], y = y1 + pomeraj[i][1];

    // ako smo na ivicama, moze da se desi da izadjemo van granica
    // lavirinta pa moramo proveriti da li smo u granicama
    // dimenzije lavirinta
    int m = lavirint.size(), n = lavirint[0].size();
    if (x < 0 || x >= m || y < 0 || y >= n)
      continue;

    // ako tu nije zid i ako to polje nije poseeno
    if (lavirint[x][y] && !posecen[x][y]) {
      // pomeramo se na to polje, belezimo da je poseceno
      posecen[x][y] = true;
      // ako od njega postoji put, onda postoji put i od polja (x1, y1)
      if (pronadjiPut(lavirint, posecen, x, y, x2, y2))
        return true;
    }
  }
  // ni u jednom od 4 moguca smera nismo uspeli da stignemo do cilja,
  // pa put ne postoji
  return false;
}
```

```
// ispituje da li postoji put kroz lavirint od polja (x1, y1) do
// (x2, y2)
bool pronadjiPut(Matrica<bool>& lavirint,
 int x1, int y1, int x2, int y2) {
  // dimenzije lavirinta
  int m = lavirint.size(), n = lavirint[0].size();

  // ni jedno polje osim starta do sada nije poseceno
  Matrica<bool> posecen = napraviMatricu(m, n, false);
  posecen[x1][y1] = true;
  // pokrecemo rekurzivnu pretragu od startnog polja
  return pronadjiPut(lavirint, posecen, x1, y1, x2, y2);
}
```

```
int main(){
  Matrica<bool> lavirint = {
    {false, false, false, false, false,
    false, false, false, false, false,
     false, false, false, false, false},

    {false, true, true, true, true,
    true,  true, true, true, true,
     true,  true, true, true, false},

    {false, true, false, false, false,
    false, false, false, false, false,
     false, false, false, false, false},

    {false, true, true, true, true,
    true,  true, true, true, true,
     true,  true, true, true, false},

    {false, false, false, false, false,
    false, false, false, false, false,
     false, false, false, true, false},

    {false, true, true, true, true,
    true,  true, true, true, true,
     true,  true, true, true, false},

    {false, true, false, false, false,
    false, false, false, false, true,
     false, false, false, false, false},

    {false, true, true, true, true,
    true,  true, true, true, true,
     true,  true, true, true, false},

    {false, false, false, false, false,
    false, false, false, false, false,
     false, false, false, false, false}};

  cout << "Put izmedju polja (1,1) i (7,13) ";
  cout << (pronadjiPut(lavirint, 1, 1, 7, 13) ? "" : "ne ");
  cout << "postoji" << endl;

  return 1;
}
```

Druga mogućnost je da se pretraga vrši *u širinu* što znači da sva polja obrađujemo u rastućem redosledu rastojanja od početnog startnog polja. Prvo je potrebno obraditi (proveriti da li se na njima nalazi cilj) sva susedna polja startnog polja, zatim sva njihova susedna polja i tako dalje. Implementaciju

vršimo pomoću reda u koji postavljamo polja koja treba obraditi. Na početku u red postavljamo samo ciljno polje. Skidamo jedan po jedan element iz reda, proveravamo da li je to ciljno polje i ako jeste, prekidamo funkciju vraćajući rezultat. Ponovo moramo voditi računa o tome da ista polja ne posećujemo više puta. Ako dodatno želimo da odredimo najkraće rastojanje od startnog do ciljnog polja, možemo čuvati pomoćnu matricu u kojoj za svako polje registrujemo to rastojanje. Za polja koja su još neposećena možemo upisati neku negativnu vrednost (npr. 1 i po tome ih raspoznavati). Kada tekući element skinemo iz reda, njegovo rastojanje od početnog polja možemo očitati iz te matrice, a zatim, za sve njegove ranije neposećene susede možemo u tu matricu upisati da im je najkraće rastojanje za jedan veće od najkraćeg rastojanja tekućeg čvora.

```
#include <iostream>
#include <vector>
#include <queue>

using namespace std;

//typedef vector<vector<bool>> Matrica;
template<typename T>
using Matrica = vector<vector<T>>;

// funkcija koja kreira matricu postavljenu na false
Matrica<int> napraviMatricu(int m, int n, int value){

  Matrica<int> mat;

  for(int i=0; i<m; i++){
    vector<int> tekuce;
    for(int j=0; j<n; j++)
      tekuce.push_back(value);

    mat.push_back(tekuce);
  }

  return mat;
}
```

```
// ispituje da li postoji put kroz lavirint od polja (x1, y1) do
// (x2, y2)
int najkraciPut(Matrica<bool>& lavirint,
int x1, int y1, int x2, int y2) {
  // dimenzije lavirinta
  int m = lavirint.size(), n = lavirint[0].size();

  // matrica u kojoj se belezi najkrace rastojanje od polja (x1, y1)
  // -1 oznacava da to rastojanje jos nije odredjeno
  Matrica<int> rastojanje = napraviMatricu(m, n, -1);
  // krecemo od polja (x1, y1) koje je samo od sebe udaljeno 0 koraka
  rastojanje[x1][y1] = 0;
  // red polja koje treba obraditi - polja se u red dodaju po
  // neopadajucem rastojanju od (x1, y1)
  queue<pair<int, int>> red;
  // pretraga krece od (x1, y1)
  red.push(make_pair(x1, y1));
  // dok se red ne isprazni
  while (!red.empty()) {
    // skidamo element iz reda
    x1 = red.front().first; y1 = red.front().second;
    red.pop();
    // citamo rastojanje od (x1, y1) do njega
    int r = rastojanje[x1][y1];
    // ako je to ciljno polje, odredili smo najkrace rastojanje
    if (x1 == x2 && y1 == y2)
      return r;
    // cetiri smera u kojima se moemo pomerati
    int pomeraj[4][2] = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};

    // pokusavamo pomeranja u svakom od tih smerova
    for (int i = 0; i < 4; i++) {
      // polje na koje se pomeramo
      int x = x1 + pomeraj[i][0], y = y1 + pomeraj[i][1];
      // proveravamo granice
      if (x < 0 || x >= m || y < 0 || y >= n)
        continue;
      // ako tu nije zid i ako to polje nije ranije poseceno
      if (lavirint[x][y] && rastojanje[x][y] == -1) {
        // odredili smo najkrae rastojanje od (x1, y1) do njega
        rastojanje[x][y] = r + 1;
        // postavljamo ga u red za dalju obradu
        red.push(make_pair(x, y));
      }
    }
  }
  return -1;
}
```

```
int main(){

  Matrica<bool> lavirint = {
    {false, false, false, false, false,
    false, false, false, false, false,

     false, false, false, false, false},

    {false, true, true, true, true,
    true,  true, true, true, true,
     true,  true, true, true, false},

    {false, true, false, false, false,
    false, false, false, false, false,
     false, false, false, false, false},

    {false, true, true, true, true,
    true,  true, true, true, true,
     true,  true, true, true, false},

    {false, false, false, false, false,
    false, false, false, false, false,
     false, false, false, true, false},

    {false, true, true, true, true,
    true,  true, true, true, true,
     true,  true, true, true, false},

    {false, true, false, false, false,
    false, false, false, false, true,
     false, false, false, false, false},

    {false, true, true, true, true,
    true,  true, true, true, true,
     true,  true, true, true, false},

    {false, false, false, false, false,
    false, false, false, false, false,
     false, false, false, false, false}};

  cout << "Duzina najkraceg puta izmedju (1,1) i (7,13) je ";
  cout << najkraciPut(lavirint, 1, 1, 7, 13) << endl;

  return 1;
}
```